National Research Council Canada
Conseil national de recherches Canada
Institute for Information Technology
Institut de Technologie de l'information
ERB-1073

NRC·CNRC

Thresholds for Object-Oriented Measures

Saida Benlarbi, Khaled El-Emam, Nishith Goel, and Shesh N. Rai
March 2000

Canada
NRC 43652

National Research Council Canada
Institute for Information Technology

Conseil national de recherches Canada
Institut de Technologie de l'information

Thresholds for Object-Oriented Measures

Saida Benlarbi, Khaled El Emam,
Nishith Goel, and Shesh N. Rai
March 2000

Copyright 2000 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

102-05443/

Thresholds for Object-Oriented Measures

Saida Benlarbi[a]
Khaled El Emam[b]
Nishith Goel[c]
Shesh Rai[d]

[a] Alcatel CID, Canada
[b] National Research Council, Canada
[c] Cistel Technology, Canada
[d] St. Jude Children's Research Hospital, USA

Abstract

A practical application of object-oriented measures is to predict which classes are likely to contain a fault. This is contended to be meaningful because object-oriented measures are believed to be indicators of psychological complexity, and classes that are more complex are likely to be faulty Recently, a cognitive theory has been proposed suggesting that there are threshold effects for many object-oriented measures. This means that object-oriented classes are easy to understand as long as their complexity is below a threshold. Above that threshold their understandability decreases rapidly, leading to an increased probability of a fault. This occurs, according to the theory, due to an overflow of short-term human memory. If this theory is confirmed, then it would provide a mechanism that would explain the introduction of faults into object-oriented systems, and would also provide some practical guidance on how to design object-oriented programs. In this paper we empirically test this theory on two C++ telecommunications systems. We test for threshold effects in a subset of the Chidamber and Kemerer (CK) suite of measures. The dependent variable was the incidence of faults that lead to field failures. Our results indicate that there are no threshold effects for any of the measures studied. This means that there is no value for the studied CK measures where the fault-proneness changes from being steady to rapidly increasing. The results are consistent across the two systems. Therefore, we can provide no support to the posited cognitive theory.

1 Introduction

Much recent research work has empirically investigated the relationship between object-oriented measures and class fault-proneness[1] [1][5][6][8][9][11][12][14][32][53]. Once validated, such measures can serve as leading indicators of fault-prone classes. Fault-prone classes can then be targeted for specific quality management action, such as more intensive inspections and testing, or they may even be redesigned.

An appealing operational approach for quality management using object-oriented measures is to develop thresholds. Thresholds are defined as [41] "heuristic values used to set ranges of desirable and undesirable metric values for measured software. These thresholds are used to identify anomalies, which may or may not be an actual problem." For example, we can say that a certain coupling measure has a threshold of seven. If the measured value for a particular class is larger than seven, then we could flag that class as high risk.

Thresholds have a practical, theoretical, and methodological significance. It is much easier for quality assurance personnel to use thresholds for identifying potentially high risk classes; they are more actionable than statistical models and equations that commonly result from validation studies. Furthermore, Hatton [33][34] has made the

1 A fault-prone class, as used in this paper, is defined as one that has a high probability of having a fault that causes a field failure. Other outcomes of interest that have been studied are productivity [17], maintenance effort [40], and development effort [17][45][44].

case for thresholds based on a cognitive theory. Specifically, he uses a human memory model to suggest that more "complex" classes will overflow short-term memory, leading to more faults. As a discipline, it is important to empirically test the versimilitude of such theories since they, if verified, can greatly improve our understanding of object-oriented design. For instance, if a threshold theory is confirmed, then it would provide one mechanism for explaining the introduction of faults into object-oriented systems. Finally, if indeed there are thresholds, then empirical models that are used to validate object-oriented measures would fit actual data much better. This would result in improved predictability of high-risk classes.

Henderson-Sellers [35] emphasizes the practical utility of thresholds[2] by stating that "An alarm would occur whenever the value of a specific internal metric exceeded some predetermined threshold." Lorenz and Kidd [41] present a number of thresholds for object-oriented measures based on their experiences with Smalltalk and C++ projects. Similarly, Rosenberg et al. [47] have developed thresholds for a number of popular object-oriented measures that are used for quality management at NASA GSFC. French [30] describes a technique for deriving thresholds, and applies it to measures collected from Ada95 and C++ programs. However, none of the above studies validated the thresholds systematically by showing that classes that exceed them are indeed more fault-prone than classes that are below the thresholds.

Chidamber et al. [17] state that the premise behind managerial use of object-oriented measures is that extreme (outlying) values signal the presence of high complexity that may require management action. They then define a lower bound for thresholds at the 80th percentile (i.e., at most 20% of the observations are considered to be above the threshold). They subsequently determine thresholds using exploratory analysis. The authors note that this is consistent with the common Pareto (80/20) heursitic. The thresholds that they use are shown to be related to managerial variables such as productivity and development effort, but they do not demonstrate such a relationship for fault-proneness.

In this paper we present a statistical technique for estimating and evaluating thresholds, and appy it on a subset of the Chidamber and Kemerer (CK) measures [15]. It has been noted that for historical reasons the CK measures are the most referenced [10]. Most commercial static analysis and measurement tools available at the time of writing also collect these measures. The study was performed initially on a C++ telecommunications applications, and replicated with another C++ telecommunications application from a different organization.

Briefly, our results indicate that there are no thresholds for any of the studied CK measures, and this is consistent across both systems. These results suggest extreme caution when developing thresholds: they must be validated because otherwise this may lead to inefficient quality management practices. Furthermore, practitioners would be prudent not to rely on published thresholds, especially those that have not been validated. From a research perspective, these results question the human memory theory that explains the existence of thresholds. To our knowledge, this theory has not received any confirmatory empirical evidence for object-oriented software, and our study constitutes strong disconfirmatory evidence.

The following section presents the justification for thresholds, an overview of thresholds already derived, and a description of the CK measures that we evaluate. In Section 3 we present the details of our research method, and Section 4 includes our results. We conclude the paper in Section 5 with a summary and the implications of our findings.

2 Background

2.1 Theory and Evidence For A Threshold Effect

A theoretical basis for developing quantitative models relating product measures and external quality measures has been provided in [9], and is summarized in Figure 1. There, it is hypothesized that the structural properties of a software component (such as its coupling) have an impact on its cognitive complexity. Cognitive complexity is defined as the mental burden of the individuals who have to deal with the component, for example, the developers, testers, inspectors, and maintainers. High cognitive complexity leads to a component exhibiting undesirable external qualities, such as increased fault-proneness and reduced maintainability.

2 The utility of thresholds for procedural applications has been noted by Lewis and Henry [39] where they describe a system that uses percentiles on procedural measures to identify potentially problematic procedures.

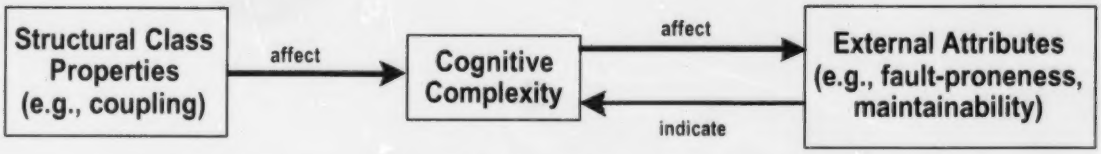

Figure 1: Theoretical basis for the development of object oriented product measures.

According to this theory, object-oriented measures that affect cognitive complexity will therefore be related with fault-proneness. Typically, structural properties such as coupling and cohesion are considered to exert significant influence on cognitive complexity. For instance, software systems composed of highly coupled classes tend to be error-prone, hard to understand and difficult to maintain. On the other hand, systems with loosely coupled classes that are highly cohesive tend to be less error prone, easier to correct, extend, and adapt to new features.

The above theory does not hypothesize any specific threshold effects. However, Hatton [33] has proposed a cognitive explanation as to why a threshold effect would exist between "complexity" measures and faults.[3] This can be considered as an extension of the above basic theory.

The proposed cognitive explanation is based on the human memory model, which consists of short-term and long-term memory. Hatton argues that Miller's work [43] shows that humans can cope with around 7 +/- 2 pieces of information at a time in short-term memory, independent of information content. He then refers to [36] where they note that the contents of long-term memory are in a coded form and the recovery codes may get scrambled under some conditions. Short-term memory incorporates a rehearsal buffer that continuously refreshes itself. He suggests that anything that can fit into short-term memory is easier to understand and less fault-prone. Pieces that are too large or too complex overflow, involving use of the more error-prone recovery code mechanism used for long-term storage. In a subsequent article, Hatton [34] extends this model to object-oriented development.

It should be noted that if a threshold theory is substantiated, this could have important implications. It would provide us with a mechanism that would explain the introduction of faults into object-oriented applications.

Below we present the empirical evidence that can be construed as supportive of this theory.

2.1.1 Size Thresholds

Hatton [34] argues that the concept of encapsulation, central to object-oriented development, lets us think about an object in isolation. If the size of this object is small enough to fit into short-term memory, then it will be easier to understand and reason about. Objects that are too large and overflow the short-term memory tend to be more fault-prone. However, a recent study demonstrated that there are no size thresholds for object-oriented classes [26]. Therefore, we do not consider size thresholds further.

2.1.2 Inheritance Thresholds

Inheritance is believed to make the understandability of object-oriented software difficult. A survey of object-oriented practitioners showed 55% of respondents agreeing that inheritance depth is a factor when attempting to understand object-oriented software [20]. It has been noted that "Inheritance gives rise to distributed class descriptions. That is, the complete description for a class D can only be assembled by examining D as well as each of D's superclasses. Because different classes are described at different places in the source code of a program (often spread across several different files), there is no single place a programmer can turn to get a complete description of a class" [38]. While this argument is stated in terms of source code, it is not difficult to generalize it to design documents. Wilde et al.'s study [57] indicated that to understand the behavior of a method one has to trace inheritance dependencies, which is considerably complicated due to dynamic binding. A similar point was made in [38] about the understandability of programs in languages that support dynamic binding, such as C++.

In a set of interviews with 13 experienced users of

3 Hatton's model also suggests that components that are of low complexity do not use short-term memory efficiently, and that failure to do so also leads to increased fault-proneness. However, this aspect of his model has been criticised recently [26] and therefore will not be considered further.

object-oriented programming, Daly et al. [19] noted that if the inheritance hierarchy is designed properly then the effect of distributing functionality over the inheritance hierarchy would not be detrimental to understanding. However, it has been argued that there exists increasing conceptual inconsistency as one travels down an inheritance hierarchy (i.e., deeper levels in the hierarchy are characterized by inconsistent extensions and/or specializations of super-classes) [23], therefore inheritance hierarchies may not be designed properly in practice. In one study Dvorak [23] found that subjects were more inconsistent in placing classes deeper in the inheritance hierarchy when compared to at higher levels in the hierarchy.

According to Hatton's theory, objects that are manipulated in short-term memory possessing inherited properties from objects already encoded in long-term memory require referencing long-term memory. However, access to long-term memory breaks the train of thought and is inherently less accurate. Therefore, according to this, it is likely that classes will be more fault-prone if they use inheritance, and this fault-proneness increases as the extent of inheritance increases. This is supported in a recent study. An experimental investigation found that making changes to a C++ program with inheritance consumed more effort than a program without inheritance, and the author attributed this to the subjects finding the inheritance program more difficult to understand based on responses to a questionnaire [13]. Another study by Cartwright and Shepperd [14] found that classes with inheritance tend to be more fault prone. This suggests that there is a threshold effect: holding everything else equal, understandability of classes is stable when there is no inheritance, but falls if there is any inheritance. In two further experiments [55] subjects were given three equivalent Java programs to make changes to, and the maintenance time was measured. One of the Java programs was 'flat', one had an inheritance depth of 3, and one had an inheritance depth of 5. The results for the first experiment indicate that the programs with inheritance depth of 3 took longer to maintain than the 'flat' program, but the program with inheritance depth of 5 took as much time as the 'flat' program. The authors attribute this to the fact that the amount of changes required to complete the maintenance task for the deepest inheritance program was smaller. The results for a second task in the first experiment and the results of the second experiment indicate that it took longer to maintain the programs with inheritance. This was attributed to the need to trace call sequences up the inheritance hierarchy in order to understand what a class is doing.

It is clear that the above studies indicate that any inheritance reduces the understandability of a class. Lorenz and Kidd [41], based on their experiences with Smalltalk and C++ projects, recommended an inheritance nesting level threshold of 6, indicating that some inheritance is not detrimental.

Exactly the opposite conclusions were obtained from another study. In [21] the authors conducted a series of classroom experiments comparing the time to perform maintenance tasks on a 'flat' C++ program and a program with three levels of inheritance. The result was a significant reduction in maintenance effort for the inheritance program. An internal replication by the same authors found the results to be in the same direction, albeit the p-value was larger. This suggests an inverse threshold effect for inheritance depth.

Clearly, the above studies suggest that there is some form of threshold effect. Although the exact value of the threshold, and even its direction, are unclear.

2.1.3 Coupling Thresholds

The object-oriented strategies of limiting a class' responsibility and reusing it in multiple contexts results in a profusion of small classes in object-oriented systems [57]. For instance, Chidamber and Kemerer [15] found in two systems studied[4] that most classes tended to have a small number of methods (0-10), suggesting that most classes are relatively simple in their construction, providing specific abstraction and functionality. Another study of three systems performed at Bellcore[5] found that half or more of the methods are fewer than four Smalltalk lines or two C++ statements, suggesting that the classes consist of small methods [57]. Many small classes means that there will be many interactions amongst the classes. This is believed to increase the complexity of the program. Wilde et al.'s [57] conclusions based on an interview-based study of two object-oriented systems at Bellcore implemented in C++ and an investigation of a PC

4 One system was developed in C++, and the other in Smalltalk.

5 The study consisted of analyzing C++ and Smalltalk systems and interviewing the developers for two of them. For a C++ system, method size was measured as the number of executable statements, and for Smalltalk size was measured by uncommented nonblank lines of code.

Smalltalk environment, all in different application domains, are concordant with this finding, in that programmers have to understand a method's context of use by tracing back through the chain of calls that reach it, and tracing the chain of methods it uses. When there are many interactions, this exacerbates the understandability problem.

Hatton's theory states when there is a diffusion of functionality, then an object in short-term memory may be referencing many objects in long-term memory. This requires retrieval (and pattern-matching in the case of polymorphism) of many other objects in long-term memory. Hence, this leads to comprehension difficulties, and according to Figure 1, greater fault-proneness. Therefore, one can argue that when the interacting objects overflow short-term memory, this will lead to an increase in fault-proneness.

2.2 Measures Studied

Below we provide a summary of the CK object-oriented measures that we study. We explicitly exclude the cohesion measure, known as LCOM, since there is no a priori reason, based on the theory above, to believe that it would exhibit a threshold effect.

2.2.1 WMC

This is the Weighted Methods per Class measure [15], and can be classified as a traditional complexity measure. It is a count of the methods in a class. The developers of this measure leave the weighting scheme as an implementation decision [15]. We weight it using cyclomatic complexity as did [40]. However, other authors did not adopt a weighting scheme [1][53]. Methods from ancestor classes are not counted and neither are "friends" in C++. This is similar to the approach taken in, for example, [1][16]. To be precise, WMC was counted after preprocessing to avoid undercounts due to macros [18].[6] Based on their experiences with object-oriented projects at NASA GSFC, Rosenberg et al. [47] define a WMC threshold of 100.

6 Note that macros embodied in #ifdef's are used to customize the implementation to a particular platform. Therefore, the method is defined at design time but its implementation is conditional on environment variables. Not counting it, as suggested in [16], would undercount methods known at design time.

2.2.2 DIT

The Depth of Inheritance Tree [15] measure is defined as the length of the longest path from the class to the root in the inheritance hierarchy. It is stated that as one goes further down the class hierarchy the more complex a class becomes, and hence more fault-prone.

The studies that present alternative thresholds for DIT have been presented above in Section 2.1.2.

2.2.3 NOC

This is the Number of Children inheritance measure [15]. This measure counts the number of classes which inherit from a particular class (i.e., the number of classes in the inheritance tree down from a class).

To our knowledge, there have been no specific thresholds specified for this measure.

2.2.4 CBO

This is the Coupling Between Object Classes coupling measure [15]. A class is coupled with another if methods of one class uses methods or attributes of the other, or vice versa. In this definition, uses can mean as a member type, parameter type, method local variable type or cast. CBO is the number of other classes to which a class is coupled. It includes inheritance-based coupling (i.e., coupling between classes related via inheritance).

Rosenberg et al. [47] have derived a threshold of 5 for CBO.

2.2.5 RFC

This is the Response for a Class coupling measure [15]. The response set of a class consists of the set Q of methods of the class, and the set of methods invoked directly by methods in Q (i.e., the set of methods that can potentially be executed in response to a message received by that class). RFC is the number of methods in the response set of the class.

Rosenberg et al. [47] derived a threshold of 100 for RFC.

2.3 Summary

The above exposition has presented the existing theoretical basis for threshold effects for object-oriented measures, and the evidence that supports it. We have also presented the CK measures that

we evaluate in the current study, as well as the thresholds that have been derived in the literature for them. In some cases, such as DIT, there are multiple conflicting thresholds.

The remainder of this paper presents a replicated study of thresholds for the CK measures. To our knowledge, this is the first attempt to empirically validate the above cognitive theory for the CK measures, and the first attempt to derive thresholds based on a systematic empirical methodology.

3 Research Method

3.1 Measurement

3.1.1 Object-Oriented Measures

The object-oriented measures described above were collected using a commercial code analyzer. In order to test for threshold effects, we also need to control for the potential confounding effect of size [24]. The size measure used was SLOC.

3.1.2 Fault Measurement

In the context of building quantitative models of software faults, it has been argued that considering faults causing field failures is a more important question to address than faults found during testing [6]. In fact, it has been argued that it is the ultimate aim of quality modeling to identify post-release fault-proneness [28]. In at least one study it was found that pre-release fault-proneness is not a good surrogate measure for post-release fault-proneness, the reason posited being that pre-release fault-proneness is a function of testing effort [29].

Therefore, faults counted for all the systems that we studied were due to field failures occuring during actual usage. For each class we characterized it as either faulty or not faulty. A faulty class had at least one fault detected during field operation. Distinct failures that are traced to the same fault are counted as a single fault.

3.2 Data Sources

Our study was performed on two C++ applications, that are described below.

3.2.1 C++ System 1

This is a telecommunications system developed in C++, and has been in operation for approximately seven years. This system has been deployed around the world in multiple sites. In total six different developers had worked on its development and evolution. It consists of 85 different classes that we analyzed.

Since the system has been evolving in functionality over the years, we selected one version for analysis where reliable fault data could be obtained.

Fault data was collected from the configuration management system. This documented the reason for each change made to the source code, and hence it was easy to identify which changes were due to faults. We focused on faults that were due to failures reported from the field. In total, 31 classes had one or more fault in them that was attributed to a field failure.

3.2.2 C++ System 2

This data set comes from a telecommunications framework written in C++ [49]. The framework implements many core design patterns for concurrent communication software. The communication software tasks provided by this framework include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, and concurrent execution and synchronization. The framework has been used in applications such as electronic medical imaging systems, configurable telecommunications systems, high-performance real-time CORBA, and web servers. Examples of its application include in the Motorola Iridium global personal communications system [51] and in network monitoring applications for telecommunications switches at Ericsson [50]. A total of 174 classes from the framework that were being reused in the development of commercial switching software constitute the system that we study. A total of 14 different programmers were involved in the development of this set of classes.

For this product, we obtained data on the faults found in the framework from actual field usage. Each fault was due to a unique field failure and represents a defect in the program that caused the failure. Failures were reported by the users of the framework. The developers of the framework documented the reasons for each delta in the version control system, and it was from this that we extracted information on whether a class was faulty. A total of 192 faults were detected in the framework at the time of writing. These faults occurred in 70

out of 174 classes.

3.3 Analysis Method

The method that we use to perform our analysis is logistic regression. Logistic regression (LR) is used to construct models when the dependent variable is binary, as in our case. The general approach we use is to construct a LR model with no threshold, and a LR model with a threshold, and then compare the two models. This is the standard technique for evaluating LR models [37].

3.3.1 Logistic Regression

The general form of an LR model is:[7]

$$\pi = \frac{1}{1+e^{-(\beta_0+\beta_1 size+\beta_2 M)}} \qquad \text{Eqn. 1}$$

where π is the probability of a class having a fault, the size variable is SLOC, and M is the specific measure that we are evaluating. The β parameters are estimated through the (unconditional)[8] maximization of a log-likelihood [37].

In the appendix we present some further diagnostics that are performed on the LR model, namely to evaluate the model, to test for collinearity, and to identify influential observations. Important methodological points to summarize here are that we use the likelihood ratio statistic, G, to test the significance of the overall model, and the R^2 value as a measure of goodness of fit. Furthermore, we compute the condition number, η, to determine whether dependencies amongst the independent variables are affecting the stability of the model. Please consult the appendix for further details.

3.3.2 Model With A Threshold

A LR model with a threshold can be defined as [54]:

$$\pi = \frac{1}{1+e^{-(\beta_0+\beta_1 size+\beta_2 (M-\tau) I_+ (M-\tau))}} \qquad \text{Eqn. 2}$$

where:

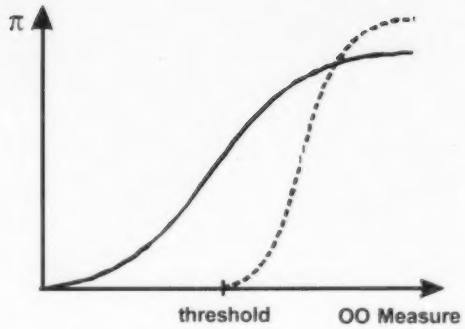

Figure 2: Relationship between the OO measure M and the probability of a fault for the threshold and no threshold models. This is the bivariate relationship assuming size is kept constant.

$$I_+(z) = \begin{cases} 0 & if\ z \le 0 \\ 1 & if\ z > 0 \end{cases} \qquad \text{Eqn. 3}$$

and τ is the measure's threshold value. In this model we keep size as a continuous variable since a previous study has indicated that there is no threshold effect for class size [26]. The difference between the no threshold and threshold model is illustrated in Figure 2. For the threshold model the probability of a fault only starts to increase once the object-oriented measure is greater than the threshold value, τ.[9]

To estimate the threshold value, τ, one can maximize the log-likelihood for the model in Eqn. 2. Ulm [54] presents an algorithm for performing this maximization.

Once a threshold is estimated, it should be evaluated. This is done by comparing the no-threshold model with the threshold model. Such a comparison is, as is typical, done using a likelihood ratio statistic (for example see [37]). The null hypothesis being tested is:

$$H_0 : \tau \le M^{(1)} = \min\ M \qquad \text{Eqn. 4}$$

where $M^{(1)}$ is the smallest value for the measure M

7 We also evaluated log and quadratic models in the logit. The conclusions were the same, and therefore we present the results for the linear model only.

8 Conditional logistic regression is used when there has been matching in the design of the study and each matched set is treated as a stratum in the analysis [7].

9 This type of model has been used in epidemiological studies, for example, to evaluate the threshold of dust concentrations in coal mines above which miners develop chronic bronchitic reactions [54]. In fact, the general approach can be applied to investigate any dose-response relationship that is postulated to have a threshold.

	Mean	Median	Std. Dev.	IQR	N>0
WMC	16.27	12	17.44	7	85
DIT	0.81	1	0.85	1	49
NOC	0.56	0	1.33	0	17
CBO	13.2	9	9.19	12	85
RFC	35.2	25	35.18	22	85
SLOC	436	280	492	244	85

Table 1: Descriptive statistics for the measures extracted from C++ System 1.

	Mean	Median	Std. Dev.	IQR	N>0
WMC	15.27	8	19.75	17	159
DIT	0.45	0	0.52	1	76
NOC	0.034	0	0.212	0	5
CBO	0.667	0	1.169	1	64
RFC	12.87	8	15.91	11	159
SLOC	64.34	41.5	61.33	56.75	174

Table 2: Descriptive statistics for the measures extracted from C++ System 2.

in the data set. If this null hypothesis is not rejected then it means that the threshold is equal to or below the minimal value. In the latter case, this is exactly like saying that the threshold model is the same as the no-threshold model. In the former case, the threshold model will be very similar to the no-threshold model since only a small proportion of the observations will have the minimal value. Hence one can conclude that there is no threshold.

The likelihood ratio statistic is computed as $2(ll(H_1)-ll(H_0))$, where ll(.) is the log-likelihood for the given model. This can be compared to a chi-square distribution with 1 degree of freedom. We use an alpha level of 0.05. Ulm [54] has performed a Monte Carlo simulation to evaluate this test and subsequently recommended its use.

It should be noted that if the estimated value for τ is equal or close to $M^{(n)}$ (the largest M value in the data set), this would mean that most of the observations in the data set would have a value of zero, making the estimated model parameters unstable. In such a case, we conclude that no threshold effect was found for this data set. Ideally, if a threshold exists then it should not be too close to the minimum or maximum values of the M measure in the data set.

4 Results

4.1 Descriptive Statistics

The descriptive statistics for the object-oriented measures and the SLOC measure are presented in Table 1 and Table 2. These include traditional summaries such as the mean and standard deviation. However, these summaries can be easily exaggerated by a single or a small number of observations. More robust analogs to these are the median and the inter-quartile range (IQR). The final column gives the number of observations that do not have zero values.

The first noticeable thing from these tables is that inheritance tends to run low in both of these systems. The NOC measure for System 2 has only five observations that are non-zero. Therefore we will not consider this variable further as it is not possible to build models when there is such little variation.

4.2 Testing For Threshold Effects

The results for the threshold models for the two systems, and the results of the comparison of the threshold and no-threshold models are shown in Table 3 and Table 4. We do not present the details of the no-threshold models since this type of validation has been reported upon elsewhere [24][25], and is not the focus of the current study.

All the threshold models have a low condition number (below the traditional threshold of 30), hence we do not consider collinearity a threat. As expected, the R^2 values run small. Counter to what one may expect, however, some of the regression coefficients are negative (e.g., for DIT for System 1, and DIT, WMC and RFC for System 2). However, these values are only different from zero by chance (i.e., if the true population value was zero, you would get a value this far negative quite frequently due to sampling variability).

For System 1, the results make clear that the CBO threshold model has a statistically significant parameter. However, the threshold model is not

Metric	G (p-value)	R^2	η	β_2 (p-value)	Threshold	Model Comparison p-value
WMC	9.17 (0.0102)	0.085	4.946	0.0946 (0.173)	11	0.72
DIT	12.21 (0.0022)	0.109	2.87	-7.498 (0.0918)	2	0.22
NOC	12.39 (0.002)	0.111	3.08	0.61 (0.0821)	2	0.602
CBO	33.64 (<0.0001)	0.301	4.71	0.1848 (<0.0001)	1	–
RFC	11.98 (0.0025)	0.107	4.5	0.0249 (0.105)	27	0.835

Table 3: Results for the threshold model for C++ System 1 and the comparison of the threshold and no-threshold models.

Metric	G (p-value)	R^2	η	β_2 (p-value)	Threshold	Model Comparison p-value
WMC	15.19 (0.0005)	0.0647	4.521	-0.03114 (0.2042)	31	0.244
DIT	15.28 (0.0005)	0.065	2.87	-6.636 (0.1919)	1	0.27
CBO	14.32 (0.0008)	0.061	2.87	5.5069 (0.3886)	8	0.393
RFC	16.15 (0.0003)	0.0697	4.43	-0.0532 (0.167)	25	0.2288

Table 4: Results for the threshold model for C++ System 2 and the comparison of the threshold and no-threshold models.

different from the no-threshold model (the last column in the tables shows the p-value for the comparison of the models). In fact for none of the measures was there a difference between the threshold and no-threshold models. The same conclusion can be drawn for System 2: none of the threshold models are different from the no-threshold models.

For the DIT measure, two different thresholds were identified for the two systems, although when we tested the null hypothesis of the threshold being equal to or less than the minimal value, it could not be rejected. The minimal value of DIT is zero, i.e., no inheritance. Therefore, this result is somewhat consistent with some of the literature mentioned earlier in that the threshold is at a DIT of zero rather than a DIT greater than zero.

Based on these results, we would conclude that the no-threshold models, which are simpler, are prefered. The addition of thresholds brings no new information.

4.3 Discussion

The results that we have presented above indicate that there is no threshold effect for a subset of the CK object-oriented measures. This means that if there is a relationship between the measure and fault-proneness, then it is a continuous one. This also means that the cognitive theory that has been postulated does not receive any support from this study, at least for object-oriented software. There is no stage on the studied CK measures where the probability of a fault changes from being steady to steadily increasing, as shown in Figure 2.

We do not claim that the existing object-oriented thresholds that have been derived from experential knowledge, such as those of Lorenz and Kidd [41] and Rosenberg et al. [47], are of no practical utility in light of our findings. Even if there is a continuous

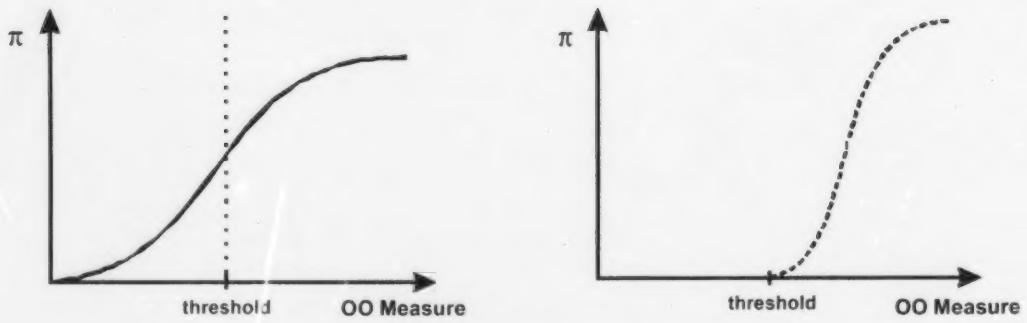

Figure 3: Different types of thresholds.

(i.e., no threshold) relationship between these measures and fault-proneness as we have found, if you draw a line at a high value of a measure and call this a threshold, classes that are above the threshold will still be the most fault-prone. This is illustrated in the left panel of Figure 3. Therefore, for the purpose of identifying the most fault-prone classes, such thresholds will likely work. But it will be noted that classes with values below the threshold can still mean high fault-proneness, just not the highest.

Had a threshold effect been identified, then classes with values below the threshold represent a "safe" region whereby designers deliberately restricting their classes within this region can have some assurance that the classes will have, everything else being equal, minimal fault-proneness. This is illustrated in the right panel of Figure 3.

4.4 Limitations

It is plausible that the two systems we studied had values on the object-oriented measures that were systematically larger than a true threshold, and hence we did not identify any threshold effects even though they exist. While the strength of this argument is diluted because it would have to be true for both systems developed by different teams in different countries, it cannot be discounted without further studies.

In our study we utilized a specific threshold model. With no prior work, this seems like a reasonable threshold model to use since it captures the theoretical claims made for threshold effects. However, We encourage other researchers to critique and improve this threshold model. Perhaps with an improved model of a threshold effect, thresholds will be identified. Therefore, while our results are clear, we do not claim that this is the last word on thresholds for object-oriented software. Rather, we hope this study will catalyze interest in object-oriented thresholds.

5 Conclusions

In a number of articles, Hatton [33][34] has posited a cognitive theory that suggests a threshold effect for many software product measures. His theory has also been extended to object-oriented software. Independently of this theory, a number of researchers have derived their own object-oriented measure thresholds based on their experience. The primary purpose of the study reported here was to test this theory empirically. We did so on two C++ telecommunications systems using a subset of the CK measures [15]. Our dependent variable was the incidence of faults that lead to field failures (fault-proneness). Our results indicated that there is no threshold effect. This was consistent across the two systems and for all the measures. While we do not claim that our results discount the threshold effects theory, the evidence is compelling against it.

The implications of these findings are twofold:

- Pracitioners deriving or using CK measure thresholds should note that classes below the thresholds are likely to still have a high fault-proneness, although perhaps not the highest fault-proneness.
- Researchers validating object-oriented measures should continue to model the relationship between object-oriented measures, at least for the CK measures, and fault-proneness using continuous assumptions (rather than threshold assumptions).

In closing, it is necessary to recognize that the formulation of theories is important for a discipline. The theories explain phenomena that we observe (i.e., they provide the mechanism). Knowledge of mechanisms can potentially lead to rapid advances in the field. Given the dearth of prior theories in software engineering, it is a brave act to articulate a theory and put it out for criticism and empirical test. Some of these theories will not survive, but others will. We therefore need to continuously propose theories, explanations, and empirically test them.

Acknowledgements

We wish to thank Anatol Kark for his comments on an earlier version of this paper.

Appendix A: Analysis Details

In this appendix we present further details on the analysis that we performed to address model diagnosis and hypothesis testing.

A.1 Diagnosing Collinearity

Collinearity is traditionally seen as being concerned with dependencies amongst independent variables. The models that we build in our study all involve size and an object-oriented measure. It is known that collinearities can exist between size and many object-oriented measures [24], and between the independent variables and the intercept [52].

Previous studies have shown that outliers can induce collinearities in regression models [42][48]. But also, it is known that collinearities may mask influential observations [2]. This has lead some authors to recommend addressing potential collinearity problems as a first step in the analysis [2], and this is the sequence that we follow.

Belsley et al. [2] propose the condition number as a collinearity diagnostic for the case of ordinary least squares regression. First, let $\hat{\mathbf{a}}_s$ be a vector of the parameter estimates, and $\mathbf{X}$ is a nx(k+1) matrix of the xij raw data, with i=1...n and j=1...k , where n is the number of observations and k is the number of independent variables. Here, the $\mathbf{X}$ matrix has a column of ones to account for the fact that the intercept is included in the models. The condition number can be obtained from the eigenvalues of the $\mathbf{X}^T\mathbf{X}$ matrix as follows:

$$\eta = \sqrt{\frac{\mu_{\max}}{\mu_{\min}}} \qquad \text{Eqn. 5}$$

where the μ's are the eigenvalues. Based on a number of experiments, Belsley et al. suggest that a condition number greater than 30 indicates mild to severe collinearity.

Belsley et al. emphasize that in the case where an intercept is included in the model, the independent variables should not be centered since this can mask the role of the constant in any near dependencies. Furthermore, the matrix must be column equilibrated, that is, each column should be scaled for equal Euclidean length. Without this, the collinearity diagnostic produces arbitrary results. Column equilibration is achieved by scaling each column in $\mathbf{X}$, $\mathbf{X_j}$, by its norm [3]:

$$\mathbf{X}_j / \|\mathbf{X}_j\| \qquad \text{Eqn. 6}$$

This diagnostic has been extended specifically to the case of LR models [22][56] by capitalizing on the analogy between the independent variable cross-product matrix in least-squares regression to the information matrix in maximum likelihood estimation, and therefore it would certainly be parsimonious to use the latter.

The information matrix in the case of LR models is [37]:

$$\hat{\mathbf{I}}(\hat{\mathbf{a}}) = \mathbf{X}^T \hat{\mathbf{V}} \mathbf{X} \qquad \text{Eqn. 7}$$

where $\hat{\mathbf{V}}$ is the diagonal matrix consisting of $\hat{\pi}_i (1 - \hat{\pi}_i)$ where $\hat{\pi}_i$ is the probability estimate from the LR model for observation i. The general approach for non-least-squares models is described by Belsley [4]. In this case, the same interpretive guidelines as for the traditional condition number are used [56].

A.2 Hypothesis Testing

The next task in evaluating the LR model is to determine whether the regression parameter is different from zero, i.e., test H_0: $\beta_1=\beta_2=0$. This can be achieved by using the likelihood ratio G statistic [37]. One first determines the log-likelihood for the model with the constant term only, and denote this l_0 for the 'null' model. Then the log-likelihood for the full model with size and the object-oriented measure

is determined, and denote this l_s. The G statistic is given by $2(l_s-l_0)$ which has a chi-square distribution with 2 degrees of freedom. To test the significance of the individual parameters, we also use a likelihood ratio statistic, whereby we compare the models with the variable and without (in this case the chi-squared statistic has only 1 degree of freedom).

In previous studies with object-oriented measures another descriptive statistic has been used, namely an R^2 statistic that is analogous to the multiple coefficient of determination in least-squares regression [11]. This is defined as $R^2=(l_0-l_s)/l_0$ and may be interpreted as the proportion of uncertainty explained by the model. We use a corrected version of this suggested by Hosmer and Lemeshow [37]. It should be recalled that this descriptive statistic will in general have low values compared to what one is accustomed to in a least-squares regression. In our study we will use the corrected R^2 statistic as a loose indicator of the quality of the LR model.

A.3 Influence Analysis

Influence analysis is performed to identify influential observations (i.e., ones that have a large influence on the LR model). This can be achieved through deletion diagnostics. For a data set with observations, estimated coefficients are recomputed times, each time deleting exactly one of the observations from the model fitting process. Pergibon has defined the $\Delta\beta$ diagnostic [46] to identify influential groups in logistic regression. The $\Delta\beta$ diagnostic is a standardized distance between the parameter estimates when a group of observations with the same x_i values is included and when they are not included in the model.

We use the $\Delta\beta$ diagnostic in our study to identify influential groups of observations. For groups that are deemed influential we investigate this to determine if we can identify substantive reasons for them being overly influential. In all cases in our study where a large $\Delta\beta$ was detected, its removal, while affecting the estimated coefficients, did not alter our conclusions.

References

[1] V. Basili, L. Briand and W. Melo: "A Validation of Object-Oriented Design Metrics as Quality Indicators". In *IEEE Transactions on Software Engineering*, 22(10):751-761, 1996.

[2] D. Belsley, E. Kuh, and R. Welsch: *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. John Wiley and Sons, 1980.

[3] D. Belsley: "A Guide to Using the Collinearity Diagnostics". In *Computer Science in Economics and Management*, 4:33-50, 1991.

[4] D. Belsley: *Conditioning Diagnostics: Collinearity and Weak Data in Regression*. John Wiley and Sons, 1991.

[5] S Benlarbi and W. Melo: "Polymorphism Measures for Early Risk Prediction". In *Proceedings of the 21st International Conference on Software Engineering*, pages 334-344, 1999.

[6] A. Binkley and S. Schach: "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Fauilures and Maintenance Measures". In *Proceedings of the 20th International Conference on Software Engineering*, pages 452-455, 1998.

[7] N. Breslow and N. Day: *Statistical Methods in Cancer Research - Volume 1 - The Analysis of Case Control Studies*, IARC, 1980.

[8] L. Briand, P. Devanbu, and W. Melo: "An Investigation into Coupling Measures for C++". In *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[9] L. Briand, J. Wuest, S. Ikonomovski, and H. Lounis: "A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study". International Software Engineering Research Network technical report ISERN-98-29, 1998.

[10] L. Briand, E. Arisholm, S. Counsell, F. Houdek, and P. Thevenod-Fosse: "Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Direction". In *Empirical Software Engineering, 4(4):387-404, 1999.*

[11] L. Briand, J. Wuest, J. Daly, and V. Porter: "Exploring the Relationships Between Design Measures and Software Quality in Object Oriented Systems". In *Journal of Systems and Software*, 51:245-273, 2000.

[12] F. Brito e Abreu and W. Melo: "Evaluating the Impact of Object-Oriented Design on Software Quality". In *Proceedings of the 3rd International Software Metrics Symposium*, pages 90-99, 1996.

[13] M. Cartwright: "An Empirical View of Inheritance". In *Information and Software Technology*, 40:795-799, 1998.

[14] M. Cartwright and M. Shepperd: "An Empirical Investigation of an Object-Oriented Software System". To appear in *IEEE Transactions on Software Engineering.*

[15] S. Chidamber and C. Kemerer: "A Metrics Suite for Object-Oriented Design". In *IEEE Transactions on Software Engineering*, 20(6):476-493, 1994.

[16] S. Chidamber and C. Kemerer: "Authors' Reply". In *IEEE Transactions on Software Engineering*, 21(3):265, 1995.

[17] S. Chidamber, D. Darcy, and C. Kemerer: "Managerial Use of Metrics for Object Oriented

Software: An Exploratory Analysis". In *IEEE Transactions on Software Engineering*, 24(8):629-639, 1998.

[18] N. Churcher and M. Shepperd: "Comments on 'A Metrics Suite for Object Oriented Design'". In *IEEE Transactions on Software Engineering*, 21(3):263-265, 1995.

[19] J. Daly, M. Wood, A. Brooks, J. Miller, and M. Roper: "Structured Interviews on the Object-Oriented Paradigm". Research Report EFoCS-7-95, Department of Computer Science, University of Strathclyde, 1995.

[20] J. Daly, J. Miller, A. Brooks, M. Roper, and M. Wood: "Issues on the Object-Oriented Paradigm: A Questionnaire Survey". Research Report EFoCS-8-95, Department of Computer Science, University of Strathclyde, 1995.

[21] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood: "Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software". In *Empirical Software Engineering*, 1(2):109-132, 1996.

[22] C. Davies, J. Hyde, S. Bangdiwala, and J. Nelson: "An Example of Dependencies Among Variables in a Conditional Logistic Regression". In S. Moolgavkar and R. Prentice (eds.): *Modern Statistical Methods in Chronic Disease Edpidemiology*. John Wiley and Sons, 1986.

[23] J. Dvorak: "Conceptual Entropy and Its Effect on Class Hierarchies". In *IEEE Computer*, pages 59-63, 1994.

[24] K. El Emam, S. Benlarbi, N. Goel, and S. Rai: "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". To appear in *IEEE Transactions on Software Engineering*, 2000.

[25] K. El Emam, S. Benlarbi, N. Goel, and S. Rai: "A Validation of Object-Oriented Metrics". Submitted for publication, 1999.

[26] K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. Rai: "The Optimal Class Size for Object-Oriented Software: A Replicated Study". Submitted for publication, 2000.

[27] N. Fenton and M. Neil: "A Critique of Software Defect Prediction Models". In *IEEE Transactions on Software Engineering*, 25(5):676-689, 1999.

[28] N. Fenton and M. Neil: "Software Metrics: Successes, Failures, and New Directions". In *Journal of Systems and Software*, 47:149-157, 1999.

[29] N. Fenton and N. Ohlsson: "Quantitative Analysis of Faults and Failures in a Complex Software System". To appear in *IEEE Transactions on Software Engineering*, 2000.

[30] V. French: "Establishing Software Metrics Thresholds". In *Proceedings of the 9th International Workshop on Software Measurement*, 1999 (available from http://www.lrgl.uqam.ca/iwsm99/index2.html).

[31] R. Harrison, L. Samaraweera, M. Dobie, and P. Lewis: "An Evaluation of Code Metrics for Object-Oriented Programs". In *Information and Software Technology*, 38:443-450, 1996.

[32] R. Harrison, S. Counsell, and R. Nithi: "Coupling Metrics for Object Oriented Design". In *Proceedings of the 5th International Symposium on Software Metrics*, pages 150-157, 1998.

[33] L. Hatton: "Re-examining the Fault Density - Component Size Connection". In *IEEE Software*, pages 89-97, 1997.

[34] L. Hatton: "Does OO Sync with How We Think ?" In *IEEE Software*, pages 46-54, May/June 1998.

[35] B. Henderson-Sellers: *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[36] E. Hilgard, R. Atkinson, and R. Atkinson: *Introduction to Psychology*. Harcourt Brace Jovanovich, 1971.

[37] D. Hosmer and S. Lemeshow: *Applied Logistic Regression*. John Wiley & Sons, 1989.

[38] M. Leijter, S. Meyers, and S. Reiss: "Support for Maintaining Object-Oriented Programs". In *IEEE Transactions on Software Engineering*, 18(12):1045-1052, 1992.

[39] J. Lewis and S. Henry: "A Methodology for Integrating Maintainability Using Software Metrics". In *Proceedings of the International Conference on Software Maintenance*, pages 32-39, 1989.

[40] W. Li and S. Henry: "Object-Oriented Metrics that Predict Maintainability". In *Journal of Systems and Software*, 23:111-122, 1993.

[41] M. Lorenz and J. Kidd: *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[42] R. Mason and R. Gunst: "Outlier-Induced Collinearities". In *Technometrics*, 27:401-407, 1985.

[43] G. Miller: "The Magical Number 7 Plus or Minus Two: Some Limits on Our Capacity for Processing Information". In *Psychological Review*, 63:81-97, 1957.

[44] V. Misic and D. Tesic: "Estimation of Effort and Complexity: An Object-oriented Case Study". In *Journal of Systems and Software*, 41:133-143, 1998.

[45] P. Nesi and T. Querci: "Effort Estimation and Prediction of Object-Oriented Systems". In *Journal of Systems and Software*, 42:89-102, 1998.

[46] D. Pergibon: "Logistic Regression Diagnostics". In *The Annals of Statistics*, 9(4):705-724, 1981.

[47] L. Rosenberg, R. Stapko, and A. Gallo: "Object-Oriented Metrics for Reliability". Presentation at *IEEE International Symposium on Software Metrics*, 1999.

[48] R. Schaefer: "Alternative Estimators in Logistic Regression when the Data are Collinear". In *The Journal of Statistical Computation and Simulation*, 25:75-91, 1986.

[49] D. Schmidt: "Using Design Patterns to Develop Reusable Object-Oriented Communication Software". In *Communications of the ACM*, 38(10):65-74, 1995.

[50] D. Schmidt and P. Stephenson: "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms". In *Proceedings of the 9th European Conference on Object Oriented Programming*, 1995.

[51] D. Schmidt: "A System of Reusable Design Patterns for Communication Software". In S. Berzuk (ed.): *The Theory and Practice of Object Systems*, Wiley, 1995.

[52] S. Simon and J. Lesage: "The Impact of Collinearity Involving the Intercept Term on the Numerical Accuracy of Regression". In *Computer Science in Economics and Management*, 1:137-152, 1988.

[53] M-H. Tang, M-H. Kao, and M-H. Chen: "An Empirical Study on Object Oriented Metrics". In *Proceedings of the Sixth International Software Metrics Symposium*, pages 242-249, 1999.

[54] K. Ulm: "A Statistical Method for Assessing A Threshold in Epidemiological Studies". In *Statistics in Medicine*, 10:341-349, 1991.

[55] B. Unger and L. Prechelt: "The Impact of Inheritance Depth on Maintenance Tasks - Detailed Description and Evaluation of Two Experiment Replications". Technical Report 19/1998, Fakultat fur Informatik, Universitaet Karlsruhe, 1998.

[56] Y. Wax: "Collinearity Diagnosis for Relative Risk Regression Analysis: An Application to Assessment of Diet-Cancer Relationship in Epidemiological Studies". In *Statistics in Medicine*, 11:1273-1287, 1992.

[57] N. Wilde, P. Matthews and R. Huitt: "Maintaining Object-Oriented Software". In *IEEE Software*, pages 75-80, Januray 1993.